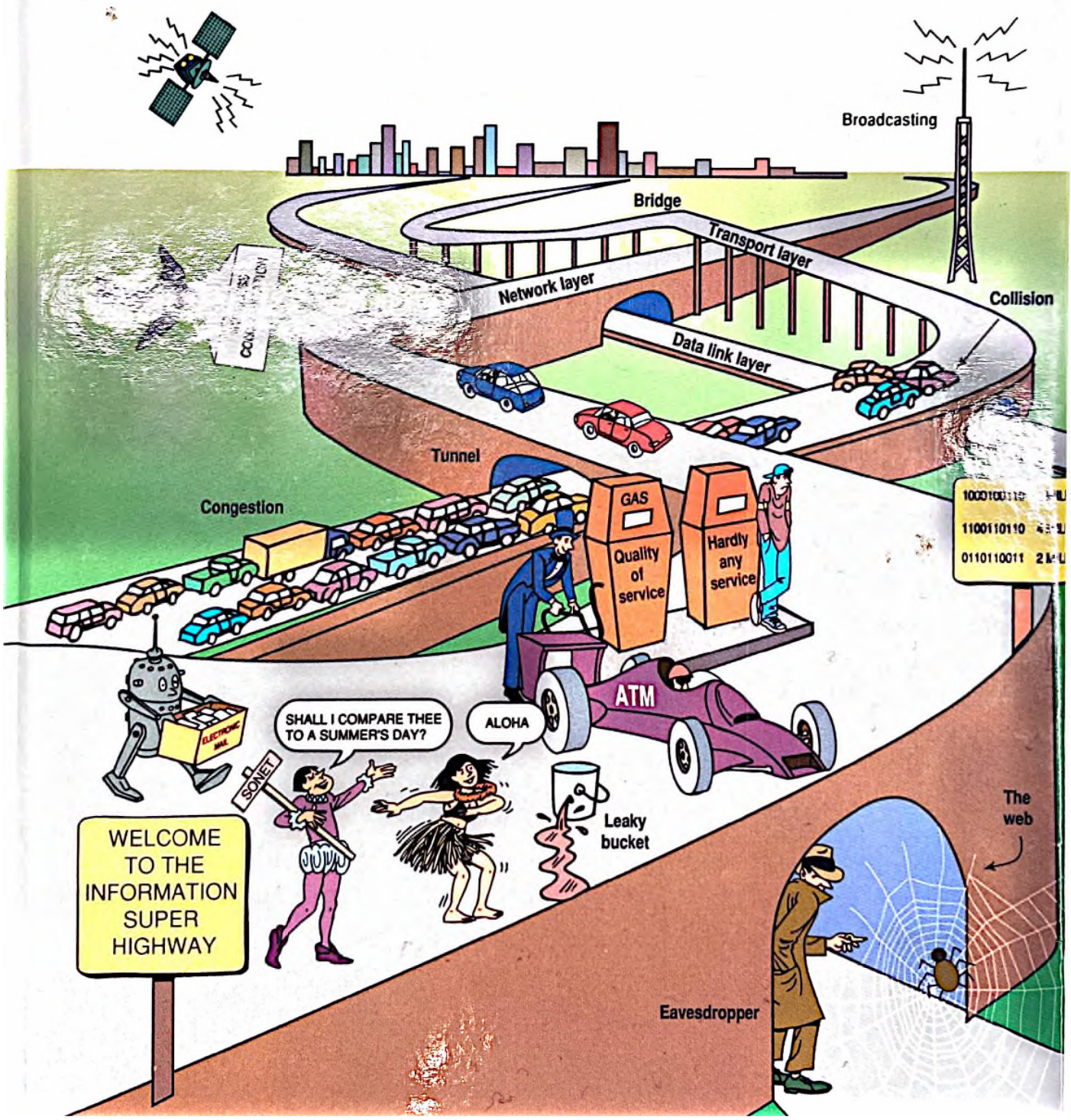
THIRD EDITION

COMPUTER NETWORKS

ANDREW S. TANENBAUM



LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 1

Computer Networks

Third Edition

Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

For book and bookstore information

http://www.prenhall.com

Prentice Hall PTR
Upper Saddle River, New Jersey 07458

Library of Congress Cataloging in Publication Data

Tanenbaum, Andrew S. 1944-.
Computer networks / Andrew S. Tanenbaum. -- 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-349945-6
1.Computer networks. I. Title.
TK5105.5.T36 1996 96-4121
004.6--dc20 CIP

Editorial/production manager: *Camille Trentacoste*
Interior design and composition: *Andrew S. Tanenbaum*
Cover design director: *Jerry Votta*
Cover designer: *Don Martinetti, DM Graphics, Inc.*
Cover concept: *Andrew S. Tanenbaum, from an idea by Marilyn Tremaine*
Interior graphics: *Hadel Studio*
Manufacturing manager: *Alexis R. Heydt*
Acquisitions editor: *Mary Franz*
Editorial Assistant: *Noreen Regina*

© 1996 by Prentice Hall PTR
Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, New Jersey 07458

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:
Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458.
Phone: (800) 382-3419; Fax: (201) 236-7141. E-mail: corpsales@prenhall.com

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

All product names mentioned herein are the trademarks of their respective owners.

Printed in the United States of America

11

ISBN 0-13-349945-6

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

7.6. THE WORLD WIDE WEB

The World Wide Web is an architectural framework for accessing linked documents spread out over thousands of machines all over the Internet. In 5 years, it went from being a way to distribute high-energy physics data to the application that millions of people think of as being "The Internet." Its enormous popularity stems from the fact that it has a colorful graphical interface that is easy for beginners to use, and it provides an enormous wealth of information on almost every conceivable subject, from aboriginals to zoology.

The Web (also known as **WWW**) began in 1989 at CERN, the European center for nuclear research. CERN has several accelerators at which large teams of scientists from the participating European countries carry out research in particle physics. These teams often have members from half a dozen or more countries. Most experiments are highly complex, and require years of advance planning and equipment construction. The Web grew out of the need to have these large teams of internationally dispersed researchers collaborate using a constantly changing collection of reports, blueprints, drawings, photos, and other documents.

The initial proposal for a web of linked documents came from CERN physicist Tim Berners-Lee in March 1989. The first (text-based) prototype was operational 18 months later. In December 1991, a public demonstration was given at the Hypertext '91 conference in San Antonio, Texas. Development continued during the next year, culminating in the release of the first graphical interface, Mosaic, in February 1993 (Vetter et al., 1994).

Mosaic was so popular that a year later, its author, Marc Andreessen left the National Center for Supercomputing Applications, where Mosaic was developed, to form a company, Netscape Communications Corp., whose goal was to develop clients, servers, and other Web software. When Netscape went public in 1995, investors, apparently thinking this was the next Microsoft, paid 1.5 billion dollars for the stock. This record was all the more surprising because the company had only one product, was operating deeply in the red, and had announced in its prospectus that it did not expect to make a profit for the foreseeable future.

In 1994, CERN and M.I.T. signed an agreement setting up the World Wide Web Consortium, an organization devoted to further developing the Web, standardizing protocols, and encouraging interoperability between sites. Berners-Lee became the director. Since then, hundreds of universities and companies have joined the consortium. M.I.T. runs the U.S. part of the consortium and the French research center, INRIA, runs the European part. Although there are more books about the Web than you can shake a stick at, the best place to get up-to-date information about the Web is (naturally) on the Web itself. The consortium's home page can be found at *http://www.w3.org* . Interested readers are referred there for links to pages covering all of the consortium's documents and activities.

In the following sections we will describe how the Web appears to the user, and, especially, how it works inside. Since the Web is basically a client-server

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 4

system, we will discuss both the client (i.e., user) side and the server side. Then we will examine the language in which Web pages are written (HTML and Java). Finally, comes an examination of how to find information on the Web.

7.6.1. The Client Side

From the users' point of view, the Web consists of a vast, worldwide collection of documents, usually just called **pages** for short. Each page may contain links (pointers) to other, related pages, anywhere in the world. Users can follow a link (e.g., by clicking on it), which then takes them to the page pointed to. This process can be repeated indefinitely, possibly traversing hundreds of linked pages while doing so. Pages that point to other pages are said to use **hypertext.**

Pages are viewed with a program called a **browser**, of which Mosaic and Netscape are two popular ones. The browser fetches the page requested, interprets the text and formatting commands that it contains, and displays the page, properly formatted, on the screen. An example is given in Fig. 7-58(a). Like many Web pages, this one starts with a title, contains some information, and ends with the email address of the page's maintainer. Strings of text that are links to other pages, called **hyperlinks**, are highlighted, either by underlining, displaying them in a special color, or both. To follow a link, the user places the cursor on the highlighted area (using the mouse or the arrow keys) and selects it (by clicking a mouse button or hitting ENTER). Although nongraphical browsers, such as Lynx, exist, they are not as popular as graphical browsers, so we will concentrate on the latter. Voice-based browsers are also being developed.

Users who are curious about the Department of Animal Psychology can learn more about it by clicking on its (underlined) name. The browser then fetches the page to which the name is linked and displays it, as shown in Fig. 7-58(b). The underlined items here can also be clicked on to fetch other pages, and so on. The new page can be on the same machine as the first one, or on a machine halfway around the globe. The user cannot tell. Page fetching is done by the browser, without any help from the user. If the user ever returns to the main page, the links that have already been followed may be shown with a dotted underline (and possibly a different color) to distinguish them from links that have not been followed. Note that clicking on the *Campus Information* line in the main page does nothing. It is not underlined, which means that it is just text and is not linked to another page.

Most browsers have numerous buttons and features to make it easier to navigate the Web. Many have a button for going back to the previous page, a button for going forward to the next page (only operative after the user has gone back from it), and a button for going straight to the user's own home page. Most browsers have a button or menu item to set a bookmark on a given page and another one to display the list of bookmarks, making it possible to revisit any of

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 5

WELCOME TO THE UNIVERSITY OF EAST PODUNK'S WWW HOME PAGE

- Campus Information
  - Admissions information
  - Campus map
  - Directions to campus
  - The UEP student body
- Academic Departments
  - Department of Animal Psychology
  - Department of Alternative Studies
  - Department of Microbiotic Cooking
  - Department of Nontraditional Studies
  - Department of Traditional Studies

Webmaster@eastpodunk.edu

(a)

THE DEPARTMENT OF ANIMAL PSYCHOLOGY

- Information for prospective majors
- Personnel
  - Faculty members
  - Graduate students
  - Nonacademic staff
- Research Projects
- Positions available
- Our most popular courses
  - Dealing with herbivores
  - Horse management
  - Negotiating with your pet
  - User-friendly doghouse construction
- Full list of courses

Webmaster@animalpsyc.eastpodunk.edu

(b)

Fig. 7-58. (a) A Web page. (b) The page reached by clicking on Department of Animal Psychology

them with a single mouse click. Pages can also be saved to disk or printed. Numerous options are generally available for controlling the screen layout and setting various user preferences. A comparison of nine browsers is given in (Berghel, 1996).

In addition to having ordinary text (not underlined) and hypertext (underlined), Web pages can also contain icons, line drawings, maps, and photographs. Each of these can (optionally) be linked to another page. Clicking on one of these elements causes the browser to fetch the linked page and display it, the same as clicking on text. With images such as photos and maps, which page is fetched next may depend on what part of the image was clicked on.

Not all pages are viewable in the conventional way. For example, some pages consist of audio tracks, video clips, or both. When hypertext pages are mixed with other media, the result is called **hypermedia**. Some browsers can display all kinds of hypermedia, but others cannot. Instead they check a configuration file to see how to handle the received data. Normally, the configuration file gives the name of a program, called an **external viewer**, or a **helper application**, to be run with the incoming page as input. If no viewer is configured, the browser usually asks the user to choose one. If no viewer exists, the user can tell the browser to save the incoming page to a disk file, or to discard it. Helper applications for producing speech are making it possible for even blind users to access the Web. Other helper applications contain interpreters for special Web languages, making it possible to download and run programs from Web pages. This mechanism makes it possible to extend the functionality of the Web itself.

Many Web pages contain large images, which take a long time to load. For example, fetching an uncompressed 640×480 (VGA) image with 24 bits per pixel (922 KB) takes about 4 minutes over a 28.8-kbps modem line. Some browsers deal with the slow loading of images by first fetching and displaying the text, then getting the images. This strategy gives the user something to read while the images are coming in and also allows the user to kill the load if the page is not sufficiently interesting to warrant waiting. An alternative strategy is to provide an option to disable the automatic fetching and display of images.

Some page writers attempt to placate potentially bored users by displaying images in a special way. First the image quickly appears in a coarse resolution. Then the details are gradually filled in. For the user, seeing the whole image after a few seconds, albeit at low resolution, is often preferable to seeing it built up slowly from the top, scan line by scan line.

Some Web pages contain forms that request the user to enter information. Typical applications of these forms are searching a database for a user-supplied item, ordering a product, or participating in a public opinion survey. Other Web pages contain maps that allow users to click on them to zoom in or get information about some geographical area. Handling forms and active (clickable) maps requires more sophisticated processing than just fetching a known page. We will describe later how these features are implemented.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 7

Some browsers use the local disk to cache pages that they have fetched. Before a page is fetched, a check is made to see if it is in the local cache. If so, it is only necessary to check if the page if still up to date. If so, the page need not be loaded again. As a result, clicking on the BACK button to see the previous page is normally very fast.

To host a Web browser, a machine must be directly on the Internet, or at least have a SLIP or PPP connection to a router or other machine that is directly on the Internet. This requirement exists because the way a browser fetches a page is to establish a TCP connection to the machine where the page is, and then send a message over the connection asking for the page. If it cannot establish a TCP connection to an arbitrary machine on the Internet, a browser will not work.

Sometimes the lengths that people will go to get Web access are amazing. At least one company is offering Web-by-Fax service. A client without Internet access calls up the Web-by-Fax server and logs in using the telephone keypad. He then types in a code identifying the Web page desired and it is faxed to the caller's fax machine.

7.6.2. The Server Side

Every Web site has a server process listening to TCP port 80 for incoming connections from clients (normally browsers). After a connection has been established, the client sends one request and the server sends one reply. Then the connection is released. The protocol that defines the legal requests and replies is called HTTP. We will study it in some detail below, but a simple example using it may provide a reasonable idea of how Web servers work. Figure 7-59 shows how the various parts of the Web model fit together.

For this example, we can imagine that the user has just clicked on some piece of text or perhaps on an icon that points to the page whose name (URL—Uniform Resource Locator) is *http://www.w3.org/hypertext/WWW/TheProject.html*. We will also explain URLs later on in this chapter. For the moment, it is sufficient to know that a URL has three parts: the name of the protocol (*http*), the name of the machine where the page is located (*www.w3.org*), and the name of the file containing the page (*hypertext/WWW/TheProject.html*). The steps that occur between the user's click and the page being displayed are as follows:

1. The browser determines the URL (by seeing what was selected).
2. The browser asks DNS for the IP address of *www.w3.org*.
3. DNS replies with 18.23.0.23.
4. The browser makes a TCP connection to port 80 on 18.23.0.23.
5. It then sends a *GET /hypertext/WWW/TheProject.html* command.
6. The *www.w3.org* server sends the file *TheProject.html*.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 8

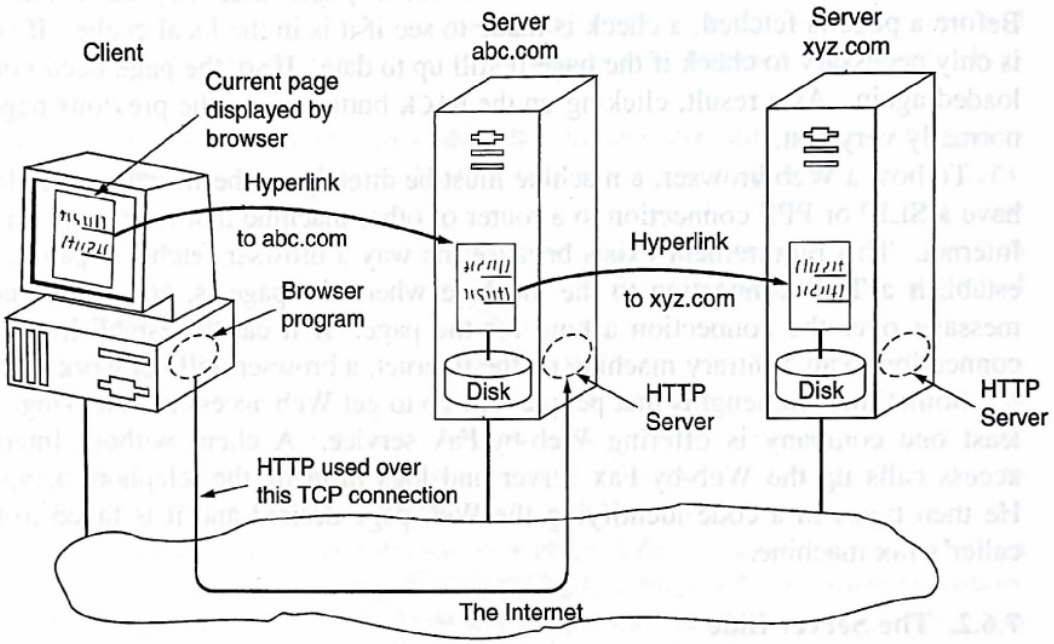


Fig. 7-59. The parts of the Web model.

7. The TCP connection is released.
8. The browser displays all the text in *TheProject.html*.
9. The browser fetches and displays all images in *TheProject.html*.

Many browsers display which step they are currently executing in a status line at the bottom of the screen. In this way, when the performance is poor, the user can see if it is due to DNS not responding, the server not responding, or simply network congestion during page transmission.

It is worth noting that for each in-line image (icon, drawing, photo, etc.) on a page, the browser establishes a new TCP connection to the relevant server to fetch the image. Needless to say, if a page contains many icons, all on the same server, establishing, using, and releasing a new connection for each one is not wildly efficient, but it keeps the implementation simple. Future revisions of the protocol will address the efficiency issue. One proposal is given in (Mogul, 1995).

Because HTTP is an ASCII protocol like SMTP, it is quite easy for a person at a terminal (as opposed to a browser) to directly talk to Web servers. All that is needed is a TCP connection to port 80 on the server. The simplest way to get such a connection is to use the Telnet program. Figure 7-60 shows a scenario of how this can be done. In this example, the lines marked *C:* are typed in by the user (client), the lines marked *T:* are produced by the Telnet program, and the lines marked *S:* are produced by the server at M.I.T.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 9

```
C: telnet  www.w3.org  80
T: Trying 18.23.0.23 ...
T: Connected to www.w3.org.
T: Escape character is '^]'.
C: GET  /hypertext/WWW/TheProject.html  HTTP/1.0
C:
    S: HTTP/1.0 200 Document follows
    S: MIME-Version: 1.0
    S: Server: CERN/3.0
    S: Content-Type: text/html
    S: Content-Length: 8247
    S:
    S: <HEAD> <TITLE> The World Wide Web Consortium (W3C) </TITLE> </HEAD>
    S: <BODY>
    S: <H1> <IMG ALIGN=MIDDLE ALT="W3C" SRC="Icons/WWW/w3c_96x67.gif">
    S: The World Wide Web Consortium </H1> <P>
    S:
    S: The World Wide Web is the universe of network-accessible information.
    S: The <A HREF="Consortium/"> World Wide Web Consortium </A>
    S: exists to realize the full potential of the Web. <P>
    S:
    S: W3C works with the global community to produce
    S: <A HREF="#Specifications"> specifications </A> and
    S: <A HREF="#Reference"> reference software </A> .
    S: W3C is funded by industrial
    S: <A HREF="Consortium/Member/List.html"> members </A>
    S: but its products are freely available to all. <P>
    S:
    S: In this document:
    S: <menu>
    S: <LI> <A HREF="#Specifications"> Web Specifications and Development Areas </A>
    S: <LI> <A HREF="#Reference"> Web Software </A>
    S: <LI> <A HREF="#Community"> The World Wide Web and the Web Community </A>
    S: <LI> <A HREF="#Joining"> Getting involved with the W3C </A>
    S: </menu>
    S: <P> <HR>
    S: <P> W3C is hosted by the
    S: <A HREF="http://www.lcs.mit.edu/"> Laboratory for Computer Science </A> at
    S: <A HREF="http://web.mit.edu/"> MIT </A> , and
    S: in Europe by <A HREF="http://www.inria.fr/"> INRIA </A> .
    S: </BODY>
```

Fig. 7-60. A sample scenario for obtaining a Web page.

Readers are encouraged to try this scenario personally (preferably from a UNIX system, because some other systems do not return the connection status). Be sure to note the spaces and the protocol version on the *GET* line, and the blank line following the *GET* line. As an aside, the actual text that will be received will differ from what is shown in Fig. 7-60 for three reasons. First, the example output here has been abridged and edited to make it fit on one page. Second, it has been cleaned up somewhat to avoid embarrassing the author, who no doubt expected thousands of people to examine the formatted page, but zero people to scrutinize the HTML that produced it. Third, the contents of the page are constantly being revised. Nevertheless, this example should give a reasonable idea of how HTTP works.

What the example shows is the following. The client, in this case a person, but normally a browser, first connects to a particular host and then sends a command asking for a particular page and specifying a particular protocol and version to use (HTTP/1.0). On line 7, the server responds with a status line telling the protocol it is using (the same as the client) and the code 200, meaning OK. This line is followed by an RFC 822 MIME message, of which five of the header lines are shown in the figure (several others have been omitted to save space). Then comes a blank line, followed by the message body. For sending a picture, the *Content-Type* field might be

```
Content-Type: Image/GIF
```

In this way, the MIME types allow arbitrary objects to be sent in a standard way. As an aside, the MIME *Content-Transfer-Encoding* header is not needed because TCP allows arbitrary byte streams, even pictures, to be sent without modification. The meaning of the commands within angle brackets used in the sample page will be discussed later in this chapter.

Not all servers speak HTTP. In particular, many older servers use the FTP, Gopher, or other protocols. Since a great deal of useful information is available on FTP and Gopher servers, one of the design goals of the Web was to make this information available to Web users. One solution is to have the browser use these protocols when speaking to an FTP or Gopher server. Some of them, in fact, use this solution, but making browsers understand every possible protocol makes them unnecessarily large.

Instead, a different solution is often used: proxy servers (Luotonen and Altis, 1994). A **proxy server** is a kind of gateway that speaks HTTP to the browser but FTP, Gopher, or some other protocol to the server. It accepts HTTP requests and translates them into, say, FTP requests, so the browser does not have to understand any protocol except HTTP. The proxy server can be a program running on the same machine as the browser, but it can also be on a free-standing machine somewhere in the network serving many browsers. Figure 7-61 shows the difference between a browser that can speak FTP and one that uses a proxy.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 11

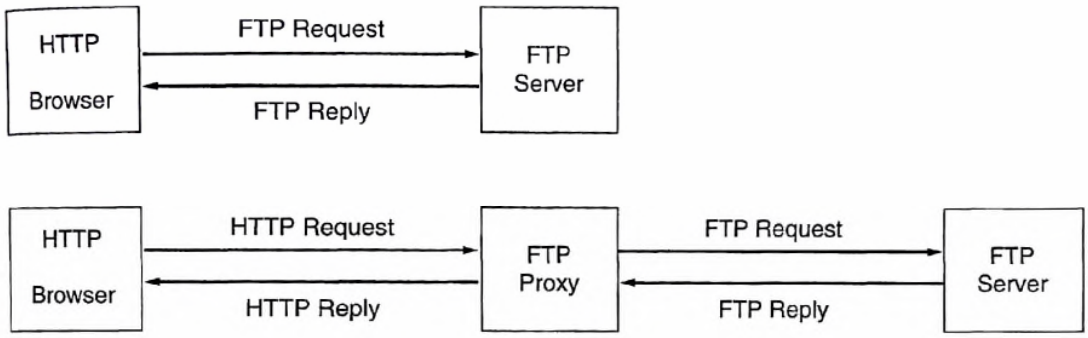


Fig. 7-61. (a) A browser that speaks FTP. (b) A browser that does not.

Often users can configure their browsers with proxies for protocols that the browsers do not speak. In this way, the range of information sources to which the browser has access is increased.

In addition to acting as a go-between for unknown protocols, proxy servers have a number of other important functions, such as caching. A caching proxy server collects and keeps all the pages that pass through it. When a user asks for a page, the proxy server checks to see if it has the page. If so, it can check to see if the page is still current. In the event that the page is still current, it is passed to the user. Otherwise, a new copy is fetched.

Finally, an organization can put a proxy server inside its firewall to allow users to access the Web, but without giving them full Internet access. In this configuration, users can talk to the proxy server, but it is the proxy server that contacts remote sites and fetches pages on behalf of its clients. This mechanism can be used, for example, by high schools, to block access to Web sites the principal feels are inappropriate for tender young minds.

For information about one of the more popular Web servers (NCSA's HTTP daemon) and its performance, see (Katz et al., 1994; and Kwan et al., 1995).

HTTP—HyperText Transfer Protocol

The standard Web transfer protocol is **HTTP (HyperText Transfer Protocol)**. Each interaction consists of one ASCII request, followed by one RFC 822 MIME-like response. Although the use of TCP for the transport connection is very common, it is not formally required by the standard. If ATM networks become reliable enough, the HTTP requests and replies could be carried in AAL 5 messages just as well.

HTTP is constantly evolving. Several versions are in use and others are under development. The material presented below is relatively basic and is unlikely to change in concept, but some details may be a little different in future versions.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 12

The HTTP protocol consists of two fairly distinct items: the set of requests from browsers to servers and the set of responses going back the other way. We will now treat each of these in turn.

All the newer versions of HTTP support two kinds of requests: simple requests and full requests. A simple request is just a single *GET* line naming the page desired, without the protocol version. The response is just the raw page, with no headers, no MIME, and no encoding. To see how this works, try making a Telnet connection to port 80 of *www.w3.org* (as shown in the first line of Fig. 7-60) and then type

```
GET /hypertext/WWW/TheProject.html
```

but without the HTTP/1.0 this time. The page will be returned with no indication of its content type. This mechanism is needed for backward compatibility. Its use will decline as browsers and servers based on full requests become standard.

Full requests are indicated by the presence of the protocol version on the *GET* request line, as in Fig. 7-60. Requests may consist of multiple lines, followed by a blank line to indicate the end of the request, which is why the blank line was needed in Fig. 7-60. The first line of a full request contains the command (of which *GET* is but one of the possibilities), the page desired, and the protocol/version. Subsequent lines contain RFC 822 headers.

Although HTTP was designed for use in the Web, it has been intentionally made more general than necessary with an eye to future object-oriented applications. For this reason, the first word on the full request line is simply the name of the **method** (command) to be executed on the Web page (or general object). The built-in methods are listed in Fig. 7-62. When accessing general objects, additional object-specific methods may also be available. The names are case sensitive, so, *GET* is a legal method but *get* is not.

Method	Description
GET	Request to read a Web page
HEAD	Request to read a Web page's header
PUT	Request to store a Web page
POST	Append to a named resource (e.g., a Web page)
DELETE	Remove the Web page
LINK	Connects two existing resources
UNLINK	Breaks an existing connection between two resources

Fig. 7-62. The built-in HTTP request methods.

The *GET* method requests the server to send the page (by which we mean object, in the most general case), suitably encoded in MIME. However, if the

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 13

GET request is followed by an *If-Modified-Since* header, the server only sends the data if it has been modified since the date supplied. Using this mechanism, a browser that is asked to display a cached page can conditionally ask for it from the server, giving the modification time associated with the page. If the cache page is still valid, the server just sends back a status line announcing that fact, thus eliminating the overhead of transferring the page again.

The *HEAD* method just asks for the message header, without the actual page. This method can be used to get a page's time of last modification, to collect information for indexing purposes, or just to test a URL for validity. Conditional *HEAD* requests do not exist.

The *PUT* method is the reverse of *GET*: instead of reading the page, it writes the page. This method makes it possible to build a collection of Web pages on a remote server. The body of the request contains the page. It may be encoded using MIME, in which case the lines following the *PUT* might include *Content-Type* and authentication headers, to prove that the caller indeed has permission to perform the requested operation.

Somewhat similar to *PUT* is the *POST* method. It too bears a URL, but instead of replacing the existing data, the new data is "appended" to it in some generalized sense. Posting a message to a news group or adding a file to a bulletin board system are examples of appending in this context. It is clearly the intention here to have the Web take over the functionality of the USENET news system.

DELETE does what you might expect: it removes the page. As with *PUT*, authentication and permission play a major role here. There is no guarantee that *DELETE* succeeds, since even if the remote HTTP server is willing to delete the page, the underlying file may have a mode that forbids the HTTP server from modifying or removing it.

The *LINK* and *UNLINK* methods allow connections to be established between existing pages or other resources.

Every request gets a response consisting of a status line, and possibly additional information (e.g., all or part of a Web page). The status line can bear the code 200 (OK), or any one of a variety of error codes, for example 304 (not modified), 400 (bad request), or 403 (forbidden).

The HTTP standards describe message headers and bodies in considerable detail. Suffice it to say that these are very close to RFC 822 MIME messages, so we will not look at them here.

7.6.3. Writing a Web Page in HTML

Web pages are written in a language called **HTML (HyperText Markup Language)**. HTML allows users to produce Web pages that include text, graphics, and pointers to other Web pages. We will begin our study of HTML with these pointers, since they are the glue that holds the Web together.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 14

URLs—Uniform Resource Locators

We have repeatedly said that Web pages may contain pointers to other Web pages. Now it is time to see how these pointers are implemented. When the Web was first created, it was immediately apparent that having one page point to another Web page required mechanisms for naming and locating pages. In particular, there were three questions that had to be answered before a selected page could be displayed:

1. What is the page called?
2. Where is the page located?
3. How can the page be accessed?

If every page were somehow assigned a unique name, there would not be any ambiguity in identifying pages. Nevertheless, the problem would not be solved. Consider a parallel between people and pages. In the United States, almost everyone has a social security number, which is a unique identifier, as no two people have the same one. Nevertheless, armed only with a social security number, there is no way to find the owner's address, and certainly no way to tell whether you should write to the person in English, Spanish, or Chinese. The Web has basically the same problems.

The solution chosen identifies pages in a way that solves all three problems at once. Each page is assigned a **URL (Uniform Resource Locator)** that effectively serves as the page's worldwide name. URLs have three parts: the protocol (also called a scheme), the DNS name of the machine on which the page is located, and a local name uniquely indicating the specific page (usually just a file name on the machine where it resides). For example, the URL for the author's department is

http://www.cs.vu.nl/welcome.html

This URL consists of three parts: the protocol (*http*), the DNS name of the host (*www.cs.vu.nl*), and the file name (*welcome.html*), with certain punctuation separating the pieces.

Many sites have certain shortcuts for file names built in. For example, *~user/* might be mapped onto *user*'s WWW directory, with the convention that a reference to the directory itself implies a certain file, say, *index.html*. Thus the author's home page can be reached at

http://www.cs.vu.nl/~ast/

even though the actual file name is different. At many sites, a null file name defaults to the organization's home page.

Now it should be clear how hypertext works. To make a piece of text clickable, the page writer must provide two items of information: the clickable text to

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 15

be displayed and the URL of the page to go to if the text is selected. When the text is selected, the browser looks up the host name using DNS. Now armed with the host's IP address, the browser then establishes a TCP connection to the host. Over that connection, it sends the file name using the specified protocol. Bingo. Back comes the page. This is precisely what we saw in Fig. 7-60.

This URL scheme is open-ended in the sense that it is straightforward to have protocols other than HTTP. In fact, URLs for various other common protocols have been defined, and many browsers understand them. Slightly simplified forms of the more common ones are listed in Fig. 7-63.

Name	Used for	Example
http	Hypertext (HTML)	http://www.cs.vu.nl/~ast/
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	/usr/suzanne/prog.c
news	News group	news:comp.os.minix
news	News article	news:AA0134223112@cs.utah.edu
gopher	Gopher	gopher://gopher.tc.umn.edu/11/Libraries
mailto	Sending email	mailto:kim@acm.org
telnet	Remote login	telnet://www.w3.org:80

Fig. 7-63. Some common URLs.

Let us briefly go over the list. The *http* protocol is the Web's native language, the one spoken by HTTP servers. It supports all the methods of Fig. 7-62, as well as whatever object-specific methods are needed.

The *ftp* protocol is used to access files by FTP, the Internet's file transfer protocol. FTP has been around more than two decades and is well entrenched. Numerous FTP servers all over the world allow people anywhere on the Internet to log in and download whatever files have been placed on the FTP server. The Web does not change this; it just makes obtaining files by FTP easier, as FTP has a somewhat arcane interface. In due course, FTP will probably vanish, as there is no particular advantage for a site to run an FTP server instead of an HTTP server, which can do everything that the FTP server can do, and more (although there are some arguments about efficiency).

It is possible to access a local file as a Web page, either by using the *file* protocol, or more simply, by just naming it. This approach is similar to using FTP but does not require having a server. Of course, it only works for local files.

The *news* protocol allows a Web user to call up a news article as though it were a Web page. This means that a Web browser is simultaneously a news reader. In fact, many browsers have buttons or menu items to make reading USENET news even easier than using standard news readers.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 16

Two formats are supported for the *news* protocol. The first format specifies a newsgroup and can be used to get a list of articles from a preconfigured news site. The second one requires the identifier of a specific news article to be given, in this case *AA0134223112@cs.utah.edu*. The browser then fetches the given article from its preconfigured news site using the NNTP protocol.

The *gopher* protocol is used by the Gopher system, which was designed at the University of Minnesota and named after the school's athletic teams, the Golden Gophers (as well as being a slang expression meaning "go for", i.e., go fetch). Gopher predates the Web by several years. It is an information retrieval scheme, conceptually similar to the Web itself, but supporting only text and no images. When a user logs into a Gopher server, he is presented with a menu of files and directories, any of which can be linked to another Gopher menu anywhere in the world.

Gopher's big advantage over the Web is that it works very well with 25×80 ASCII terminals, of which there are still quite a few around, and because it is text based, it is very fast. Consequently, there are thousands of Gopher servers all over the world. Using the *gopher* protocol, Web users can access Gopher and have each Gopher menu presented as a clickable Web page. If you are not familiar with Gopher, try the example given in Fig. 7-63 or have your favorite Web search engine look for "gopher."

Although the example given does not illustrate it, it is also possible to send a complete query to a Gopher server using the *gopher+* protocol. What is displayed is the result of querying the remote Gopher server.

The last two protocols do not really have the flavor of fetching Web pages, and are not supported by all browsers, but are useful anyway. The *mailto* protocol allows users to send email from a Web browser. The way to do this is to click on the OPEN button and specify a URL consisting of *mailto:* followed by the recipient's email address. Most browsers will respond by popping up a form containing slots for the subject and other header lines and space for typing the message.

The *telnet* protocol is used to establish an on-line connection to a remote machine. It is used the same way as the Telnet program, which is not surprising, since most browsers just call the Telnet program as a helper application. As an exercise, try the scenario of Fig. 7-60 again, but now using a Web browser.

In short, the URLs have been designed to not only allow users to navigate the Web, but to deal with FTP, news, Gopher, email, and telnet as well, making all the specialized user interface programs for those other services unnecessary, and thus integrating nearly all Internet access into a single program, the Web browser. If it were not for the fact that this scheme was designed by a physics researcher, it could easily pass for the output of some software company's advertising department.

Despite all these nice properties, the growing use of the Web has turned up an inherent weakness in the URL scheme. A URL points to one specific host. For

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 17

pages that are heavily referenced, it is desirable to have multiple copies far apart, to reduce the network traffic. The trouble is that URLs do not provide any way to reference a page without simultaneously telling where it is. There is no way to say: "I want page xyz, but I do not care where you get it." To solve this problem and make it possible to replicate pages, the IETF is working on a system of **URI**s (**Universal Resource Identifiers**). A URI can be thought of as a generalized URL. This topic is the subject of much current research.

Although we have discussed only absolute URLs here, relative URLs also exist. The difference is analogous to the difference between the absolute file name */usr/ast/foobar* and just *foobar* when the context is unambiguously defined.

HTML—HyperText Markup Language

Now that we have a good idea of how URLs work, it is time to look at HTML itself. HTML is an application of ISO standard 8879, **SGML (Standard Generalized Markup Language)**, but specialized to hypertext and adapted to the Web.

As mentioned earlier, HTML is a markup language, a language for describing how documents are to be formatted. The term "markup" comes from the old days when copyeditors actually marked up documents to tell the printer—in those days, a human being—which fonts to use, and so on. Markup languages thus contain explicit commands for formatting. For example, in HTML, **<B> means start boldface mode, and** </B> means leave boldface mode. The advantage of a markup language over one with no explicit markup is that writing a browser for it is straightforward: the browser simply has to understand the markup commands. TeX and troff are other well-known examples of markup languages.

Documents written in a markup language can be contrasted to documents produced with a WYSIWYG (What You See Is What You Get) word processor, such as MS-Word® or WordPerfect®. These systems may store their files with hidden embedded markup so they can reproduce them later, but not all of them work this way. Word processors for the Macintosh, for example, keep the formatting information in separate data structures, not as commands embedded in the user files.

By embedding the markup commands within each HTML file and standardizing them, it becomes possible for any Web browser to read and reformat any Web page. Being able to reformat Web pages after receiving them is crucial because a page may have been produced full screen on a 1024×768 display with 24-bit color but may have to be displayed in a small window on a 640×480 screen with 8-bit color. Proprietary WYSIWYG word processors cannot be used on the Web because their internal markup languages (if any) are not standardized across vendors, machines and operating systems. Also, they do not handle reformatting for different-sized windows and different resolution displays. However, word processing programs can offer the option of saving documents in HTML instead of in the vendor's proprietary format, and some of them already do.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 18

Like HTTP, HTML is in a constant state of flux. When Mosaic was the only browser, the language it interpreted, HTML 1.0, was the de facto standard. When new browsers came along, there was a need for a formal Internet standard, so the HTML 2.0 standard was produced. HTML 3.0 was initially created as a research effort to add many new features to HTML 2.0, including tables, toolbars, mathematical formulas, advanced style sheets (for defining page layout and the meaning of symbols), and more.

The official standardization of HTML is being managed by the WWW Consortium, but various browser vendors have added their own ad hoc extensions. These vendors hope to get people to write Web pages using their extensions, so readers of these pages will need the vendor's browser to properly interpret the pages. This tendency does not make HTML standardization any easier.

Below we will give a brief introduction to HTML, just to give an idea of what it is like. While it is certainly possible to write HTML documents with any standard editor, and many people do, it is also possible to use special HTML editors that do most of the work (but correspondingly give the user less control over all the details of the final result).

A proper Web page consists of a head and a body enclosed by <HTML> and </HTML> **tags** (formatting commands), although most browsers do not complain if these tags are missing. As can be seen from Fig. 7-64(a), the head is bracketed by the <HEAD> and </HEAD> tags and the body is bracketed by the <BODY> and </BODY> tags. The commands inside the tags are called **directives**. Most HTML tags have this format, that is, <SOMETHING> to mark the beginning of something and </SOMETHING> to mark its end. Numerous other examples of HTML are easily available. Most browsers have a menu item VIEW SOURCE or something like that. Selecting this item displays the current page's HTML source, instead of its formatted output.

Tags can be in either lowercase or uppercase. Thus <HEAD> and <head> mean the same thing, but the former stands out better for human readers. Actual layout of the HTML document is irrelevant. HTML parsers ignore extra spaces and carriage returns since they have to reformat the text to make it fit the current display area. Consequently, white space can be added at will to make HTML documents more readable, something most of them are badly in need of. As another consequence, blank lines cannot be used to separate paragraphs, as they are simply ignored. An explicit tag is required.

Some tags have (named) parameters. For example

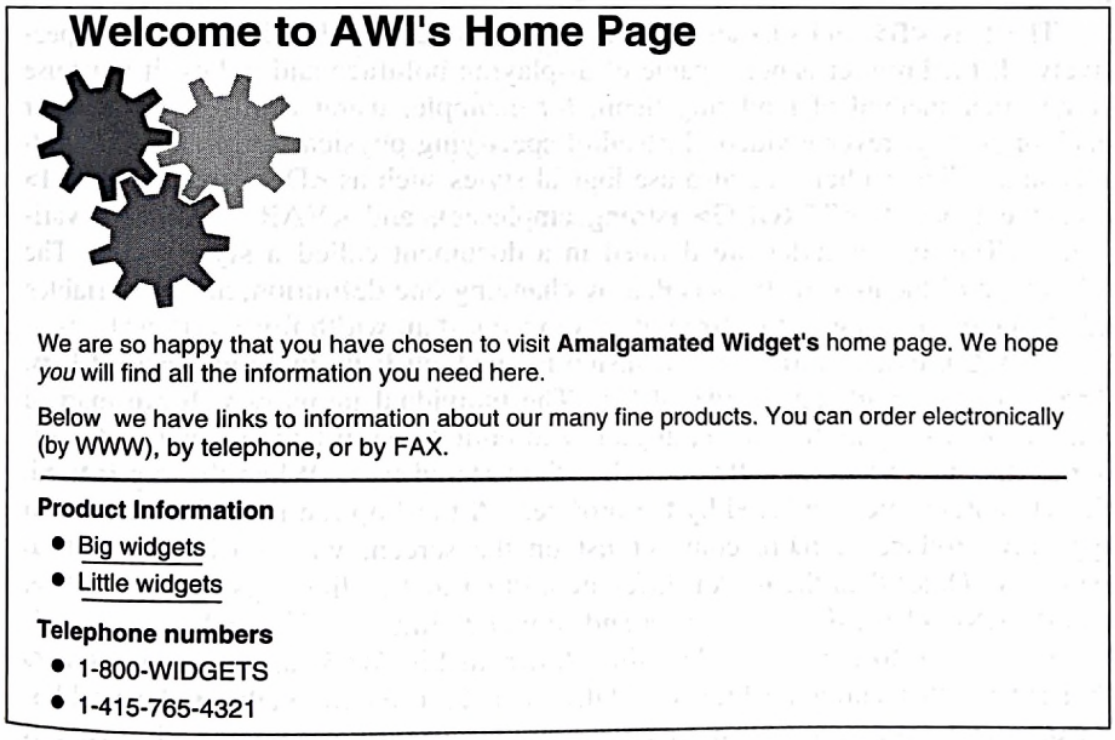
```
<IMG SRC="abc" ALT="foobar">
```

is a tag, <IMG>, with parameter *SRC* set equal to *abc* and parameter *ALT* set equal to *foobar*. For each tag, the HTML standard gives a list of what the permitted parameters, if any, are, and what they mean. Because each parameter is named, the order in which the parameters are given is not significant.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 19

```
<HTML> <HEAD> <TITLE> AMALGAMATED WIDGET, INC. </TITLE> </HEAD>
<BODY> <H1> Welcome to AWI's Home Page </H1>
<IMG SRC="http://www.widget.com/images/logo.gif" ALT="AWI Logo"> <BR>
We are so happy that you have chosen to visit <B> Amalgamated Widget's</B>
home page. We hope <I> you </I> will find all the information you need here.
<P>Below we have links to information about our many fine products.
You can order electronically (by WWW), by telephone, or by fax. <HR>
<H2> Product information </H2>
<UL> <LI> <A HREF="http://widget.com/products/big">  Big widgets   </A>
     <LI> <A HREF="http://widget.com/products/little"> Little widgets </A>
</UL>
<H2> Telephone numbers </H2>
<UL> <LI> By telephone: 1-800-WIDGETS
     <LI> By fax: 1-415-765-4321
</UL> </BODY> </HTML>
```

(a)



(b)

Fig. 7-64. (a) The HTML for a sample Web page. (b) The formatted page.

Technically, HTML documents are written in the ISO 8859-1 Latin-1 character set, but for users whose keyboards only support ASCII, escape sequences are present for the special characters, such as è. The list of special characters is given in the standard. All of them begin with an ampersand and end with a semicolon. For example, è produces è and é produces é. Since <, >, and & have special meanings, they can be expressed only with their escape sequences, < > and & respectively.

The main item in the head is the title, delimited by <TITLE> and </TITLE>, but certain kinds of meta-information may also be present. The title itself is not displayed on the page. Some browsers use it to label the page's window.

Let us now take a look at some of the other features illustrated in Fig. 7-64. All of the tags used in Fig. 7-64 and some others are shown in Fig. 7-65. Headings are generated by an <H*n*> tag, where *n* is a digit in the range 1 to 6. <H1> is the most important heading; <H6> is the least important one. It is up to the browser to render these appropriately on the screen. Typically the lower numbered headings will be displayed in a larger and heavier font. The browser may also choose to use different colors for each level of heading. Typically <H1> headings are large and boldface with at least one blank line above and below. In contrast, <H2> headings are in a smaller font, and with less space above and below.

The tags <B> and <I> are used to enter boldface and italics mode, respectively. If the browser is not capable of displaying boldface and italics, it must use some other method of rendering them, for example, using a different color for each or perhaps reverse video. Instead of specifying physical styles such as boldface and italics, authors can also use logical styles such as <DN> (define), <EM> (weak emphasis), <STRONG> (strong emphasis), and <VAR> (program variables). The logical styles are defined in a document called a **style sheet**. The advantage of the logical styles is that by changing one definition, all the variables can be changed, for example, from italics to a constant width font.

HTML provides various mechanisms for making lists, including nested lists. The <UL> tag starts an unordered list. The individual items, which are marked with the <LI> tag in the source, appear with bullets (•) in front of them. A variant of this mechanism is <OL>, which is for ordered lists. When this tag is used, the <LI> items are numbered by the browser. A third option is <MENU>, which typically produces a more compact list on the screen, with no bullets and no numbers. Other than the use of different starting and ending tags, <UL>, <OL>, and <MENU> have the same syntax and similar results.

In addition to the list mechanisms shown in Fig. 7-65, there are two others that are worth mentioning briefly. <DIR> can be used for making short tables. Also, <DL> and </DL> can make definition lists (glossaries) with two-part entries, whose parts are defined by <DT> and <DD> respectively. The first is for the name, the second for its meaning. These features are largely superseded by the (more general and complex) table mechanism, described below.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 21

Tag	Description
<HTML> ... </HTML>	Declares the Web page to be written in HTML
<HEAD> ... </HEAD>	Delimits the page's head
<TITLE> ... </TITLE>	Defines the title (not displayed on the page)
<BODY> ... </BODY>	Delimits the page's body
<H*n*> ... </H*n*>	Delimits a level *n* heading
<B> ... </B>	Set ... in boldface
<I> ... </I>	Set ... in italics
<UL> ... </UL>	Brackets an unordered (bulleted) list
<OL> ... </OL>	Brackets a numbered list
<MENU> ... </MENU>	Brackets a menu of <LI> items
<LI>	Start of a list item (there is no </LI>)
 	Force a break here
<P>	Start of paragraph
<HR>	Horizontal rule
<PRE> ... </PRE>	Preformatted text; do not reformat
<IMG SRC="...">	Load an image here
<A HREF="..."> ... </A>	Defines a hyperlink

Fig. 7-65. A selection of common HTML tags. Some have additional parameters.

The <BR>, <P>, and <HR> tags all indicate a boundary between sections of text. The precise format can be determined by the style sheet associated with the page. The <BR> tag just forces a line break. Typically, browsers do not insert a blank line after <BR>. In contrast, <P> starts a paragraph, which might, for example, insert a blank line and possibly some indentation. (Theoretically, </P> exists to mark the end of a paragraph, but it is rarely used; most HTML authors do not even know it exists.) Finally, <HR> forces a break and draws a horizontal line across the screen.

HTML 1.0 had no ability to display tables or other formatted information. Worse yet, if the HTML writer carefully formatted a table by judicious use of spaces and carriage returns, browsers would ignore all the layout and display the page as if all the formatted material were unformatted. To prevent browsers from messing up carefully laid out text, the <PRE> and </PRE> tags were provided. They are instructions to the browser to just display everything in between literally, character for character, without changing anything. As the table and other fancy layout features become more widely implemented, the need for <PRE> will

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 22

diminish, except for program listings, for which most programmers will tolerate no formatting other than their own.

HTML allows images to be included in-line on a Web page. The <IMG> tag specifies that an image is to be loaded at the current position in the page. It can have several parameters. The *SRC* parameter gives the URL (or URI) of the image. The HTML standard does not specify which graphic formats are permitted. In practice, all browsers support GIF files and many support JPEG files as well. Browsers are free to support other formats, but this extension is a two-edged sword. If a user is accustomed to a browser that supports, say, BMP files, he may include these in his Web pages and later be surprised when other browsers just ignore all of his wonderful art.

Other parameters of <IMG> are *ALIGN*, which controls the alignment of the image with respect to the text baseline (*TOP*, *MIDDLE*, *BOTTOM*), *ALT*, which provides text to use instead of the image when the user has disabled images, and *ISMAP*, a flag indicating that the image is an active map.

Finally, we come to hyperlinks, which use the <A> (anchor) and </A> tags. Like <IMG>, <A> has various parameters, including *HREF* (the URL), *NAME* (the hyperlink's name), and *METHODS* (access methods), among others. The text between the <A> and </A> is displayed. If it is selected, the hyperlink is followed to a new page. It is also permitted to put an <IMG> image there, in which case clicking on the image also activates the hyperlink.

As an example, consider the following HTML fragment:

```
<A HREF="http://www.nasa.gov"> NASA's home page </A>
```

When a page with this fragment is displayed, what appears on the screen is

NASA's home page

If the user subsequently clicks on this text, the browser immediately fetches the page whose URL is *http://www.nasa.gov* and displays it.

As a second example, now consider

```
<A HREF="http://www.nasa.gov"> <IMG SRC="shuttle.gif" ALT="NASA"> </A>
```

When displayed, this page shows a picture (e.g., of the space shuttle). Clicking on the picture switches to NASA's home page, just as clicking on the underlined text did in the previous example. If the user has disabled automatic image display, the text NASA will be displayed where the picture belongs.

The <A> tag can take a parameter *NAME* to plant a hyperlink, so it can be referred to from within the page. For example, some Web pages start out with a clickable table of contents. By clicking on an item in the table of contents, the user jumps to the corresponding section of the page.

One feature that HTML 2.0 did not include and which many page authors missed, was the ability to create tables whose entries could be clicked on to active hyperlinks. As a consequence, a large amount of work was done to add tables to

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 23

HTML 3.0. Below we give a very brief introduction to tables, just to capture the essential flavor.

An HTML table consists of one or more rows, each consisting of one or more **cells**. Cells can contain a wide range of material, including text, figures, and even other tables. Cells can be merged, so, for example, a heading can span multiple columns. Page authors have limited control over the layout, including alignment, border styles, and cell margins, but the browsers have the final say in rendering tables.

An HTML table definition is listed in Fig. 7-66(a) and a possible rendition is shown in Fig. 7-66(b). This example just shows a few of the basic features of HTML tables. Tables are started by the <TABLE> tag. Additional information can be provided to describe general properties of the table.

The <CAPTION> tag can be used to provide a figure caption. Each row is started with a <TR> (Table Row) tag. The individual cells are marked as <TH> (Table Header) or <TD> (Table Data). The distinction is made to allow browsers to use different renditions for them, as we have done in the example.

Numerous other tags are also allowed in tables. They include ways to specify horizontal and vertical cell alignments, justification within a cell, borders, grouping of cells, units, and more.

Forms

HTML 1.0 was basically one way. Users could call up pages from information providers, but it was difficult to send information back the other way. As more and more commercial organizations began using the Web, there was a large demand for two-way traffic. For example, many companies wanted to be able to take orders for products via their Web pages, software vendors wanted to distribute software via the Web and have customers fill out their registration cards electronically, and companies offering Web searching wanted to have their customers be able to type in search keywords.

These demands led to the inclusion of **forms** starting in HTML 2.0. Forms contain boxes or buttons that allow users to fill in information or make choices and then send the information back to the page's owner. They use the <INPUT> tag for this purpose. It has a variety of parameters for determining the size, nature, and usage of the box displayed. The most common forms are blank fields for accepting user text, boxes that can be checked, active maps, and SUBMIT buttons. The example of Fig. 7-67 illustrates some of these choices.

Let us start our discussion of forms by going over this example. Like all forms, this one is enclosed between the <FORM> and </FORM> tags. Text not enclosed in a tag is just displayed. All the usual tags (e.g., <B>) are allowed in a form. Three kinds of input boxes are used in this form.

The first kind of input box follows the text "Name". The box is 46 characters wide and expects the user to type in a string, which is then stored in the variable

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 24

```
<HTML> <HEAD> <TITLE> A sample page with a table </TITLE> </HEAD>
<BODY>
<TABLE BORDER=ALL RULES=ALL>
<CAPTION> Some Differences between HTML Versions </CAPTION>
<COL ALIGN=LEFT>
<COL ALIGN=CENTER>
<COL ALIGN=CENTER>
<COL ALIGN=CENTER>
<TR> <TH>Item  <TH>HTML 1.0  <TH>HTML 2.0  <TH>HTML 3.0
<TR> <TH> Active Maps and Images <TD> <TD> x <TD> x
<TR> <TH> Equations <TD> <TD> <TD> x
<TR> <TH> Forms <TD> <TD> x <TD> x
<TR> <TH> Hyperlinks x <TD> <TD> x <TD> x
<TR> <TH> Images <TD> x <TD> x <TD> x
<TR> <TH> Lists <TD> x <TD> x <TD> x
<TR> <TH> Toolbars <TD> <TD> <TD> x
<TR> <TH> Tables <TD> <TD> <TD> x
</TABLE> </BODY> </HTML>
```

(a)

Some Differences between HTML Versions

Item	HTML 1.0	HTML 2.0	HTML 3.0
Active Maps and Images		x	x
Equations			x
Forms		x	x
Hyperlinks	x	x	x
Images	x	x	x
Lists	x	x	x
Toolbars			x
Tables			x

Fig. 7-66. (a) An HTML table. (b) A possible rendition of this table.

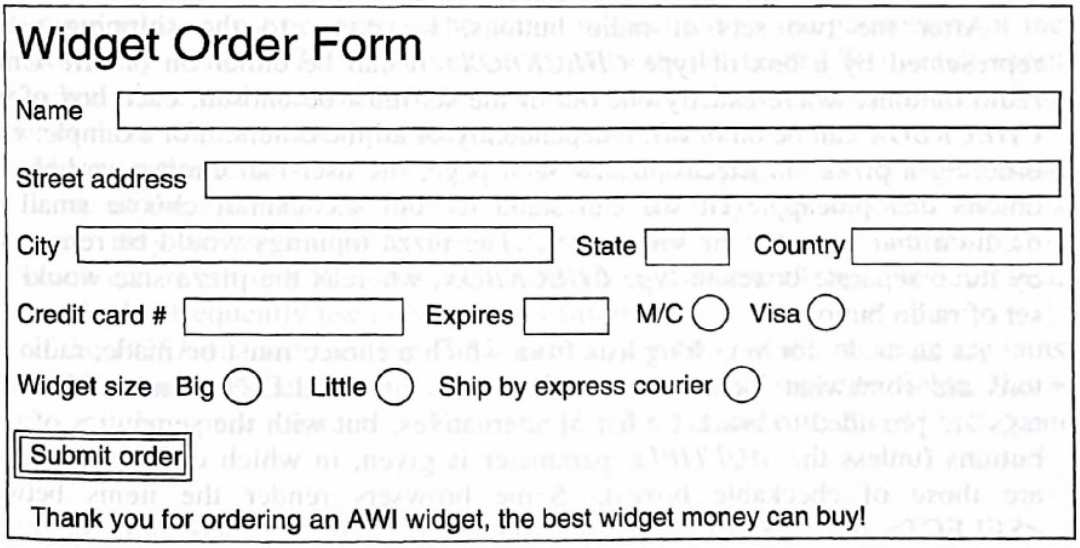
customer for later processing. The <P> tag instructs the browser to display subsequent text and boxes on the next line, even if there is room on the current line. By using <P> and other layout tags, the author of the page can control the look of the form on the screen.

The next line of the form asks for the user's street address, 40 columns wide, also on a line by itself. Then comes a line asking for the city, state, and country. No <P> tags are used between the fields here, so the browser displays them all on one line if they will fit. As far as the browser is concerned, this paragraph just contains six items: three strings alternating with three boxes. It displays them linearly from left to right, going over to a new line whenever the current line

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 25

```
<HTML> <HEAD> <TITLE> AWI CUSTOMER ORDERING FORM </TITLE> </HEAD>
<BODY>
<H1> Widget Order Form </H1>
<FORM ACTION="http://widget.com/cgi-bin/widgetorder" METHOD=POST>
Name <INPUT NAME="customer" SIZE=46> <P>
Street Address <INPUT NAME="address" SIZE=40> <P>
City <INPUT NAME="city" SIZE=20> State <INPUT NAME="state" SIZE =4>
Country <INPUT NAME="country" SIZE=10> <P>
Credit card # <INPUT NAME="cardno" SIZE=10>
Expires <INPUT NAME="expires" SIZE=4>
M/C <INPUT NAME="cc" TYPE=RADIO VALUE="mastercard">
VISA <INPUT NAME="cc" TYPE=RADIO VALUE="visacard"> <P>
Widget size Big <INPUT NAME="product" TYPE=RADIO VALUE="expensive">
Little <INPUT NAME="product" TYPE=RADIO VALUE="cheap">
Ship by express courier <INPUT NAME="express" TYPE=CHECKBOX> <P>
<INPUT TYPE=SUBMIT VALUE="Submit order"> <P>
Thank you for ordering an AWI widget, the best widget money can buy!
</FORM> </BODY> </HTML>
```

(a)



(b)

Fig. 7-67. (a) The HTML for an order form. (b) The formatted page.

cannot hold the next item. Thus it is conceivable that on a 1024×768 screen all three strings and their corresponding boxes will appear on the same line, but on a 640×480 screen they might be split over two lines. In the worst scenario, the word "Country" is at the end of one line and its box is at the beginning of the next line. There is no way to tell the browser to force the box adjacent to the text.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 26

The next line asks for the credit card number and expiration date. Transmitting credit card numbers over the Internet should only be done when adequate security measures have been taken. For example, some, but not all, browsers encrypt information sent by users. Even then, secure communication and key management are complicated matters and are subject to many kinds of attacks, as we saw earlier.

Following the expiration date we encounter a new feature: radio buttons. These are used when a choice must be made among two or more alternatives. The intellectual model here is a car radio with half a dozen buttons for choosing stations. The browser displays these boxes in a form that allows the user to select and deselect them by clicking on them (or using the keyboard). Clicking on one of them turns off all the other ones in the same group. The visual presentation depends on the graphical interface being used. It is up to the browser to choose a form that is consistent with Windows, Motif, OS/2 Warp, or whatever windowing system is being used. The widget size also uses two radio buttons. The two groups are distinguished by their *NAME* field, not by static scoping using something like <RADIOBUTTON> ... </RADIOBUTTON>.

The *VALUE* parameters are used to indicate which radio button was pushed. Depending on which of the credit card options the user has chosen, the variable *cc* will be set to either the string "mastercard" or the string "visacard".

After the two sets of radio buttons, we come to the shipping option, represented by a box of type *CHECKBOX*. It can be either on or off. Unlike radio buttons, where exactly one out of the set must be chosen, each box of type *CHECKBOX* can be on or off, independently of all the others. For example, when ordering a pizza via Electropizza's Web page, the user can choose sardines *and* onions *and* pineapple (if she can stand it), but she cannot choose small *and* medium *and* large for the same pizza. The pizza toppings would be represented by three separate boxes of type *CHECKBOX*, whereas the pizza size would be a set of radio buttons.

As an aside, for very long lists from which a choice must be made, radio buttons are somewhat inconvenient. Therefore, the <SELECT> and </SELECT> tags are provided to bracket a list of alternatives, but with the semantics of radio buttons (unless the *MULTIPLE* parameter is given, in which case the semantics are those of checkable boxes). Some browsers render the items between <SELECT> and </SELECT> as a pop-up menu.

We have now seen two of the built-in types for the <INPUT> tag: *RADIO* and *CHECKBOX*. In fact, we have already seen a third one as well: *TEXT*. Because this type is the default, we did not bother to include the parameter *TYPE = TEXT*, but we could have. Two other types are *PASSWORD* and *TEXTAREA*. A *PASSWORD* box is the same as a *TEXT* box, except that the characters are not displayed as they are typed. A *TEXTAREA* box is also the same as a *TEXT* box, except that it can contain multiple lines.

Getting back to the example of Fig. 7-67, we now come across an example of

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 27

a *SUBMIT* button. When this is clicked, the user information on the form is sent back to the machine that provided the form. Like all the other types, *SUBMIT* is a reserved word that the browser understands. The *VALUE* string here is the label on the button and is displayed. All boxes can have values; we only needed that feature here. For *TEXT* boxes, the contents of the *VALUE* field are displayed along with the form, but the user can edit or erase it. *CHECKBOX* and *RADIO* boxes can also be initialized, but with a field called *CHECKED* (because *VALUE* just gives the text, but does not indicate a preferred choice).

The browser also understands the *RESET* button. When clicked, it resets the form to its initial state.

Two more types are worth noting. The first is the *HIDDEN* type. This is output only; it cannot be clicked or modified. For example, when working through a series of pages throughout which choices have to be made, previously made choices might be of *HIDDEN* type, to prevent them from being changed.

Our last type is *IMAGE*, which is for active maps (and other clickable images). When the user clicks on the map, the (x, y) coordinates of the pixel selected (i.e., the current mouse position) are stored in variables and the form is automatically returned to the owner for further processing.

Forms can be submitted in three ways: using the submit button, clicking on an active map, or typing ENTER on a one-item *TEXT* form. When a form is submitted, some action must be taken. The action is specified by the parameters of the <FORM> tag. The *ACTION* parameter specifies the URL (or URI) to tell about the submission, and the *METHOD* parameter tells which method to use. The order of these (and all other) parameters is not significant.

The way the form's variables are sent back to the page's owner depends on the value of the *METHOD* parameter. For *GET*, the only way to return values is to cheat: they are appended to the URL, separated by a question mark. This approach can result in URLs that are thousands of characters long. Nevertheless, this method is frequently used because it is simple.

If the *POST* method (see Fig. 7-62) is used, the body of the message contains the form's variables and their values. The & is used to separate fields; the + represents the space character. For example, the response to the widget form might be

```
customer=John+Doe&address=100+Main+St.&city=White+Plains&
state=NY&country=USA&cardno=1234567890&expires=6/98&cc=mastercard&
product=cheap&express=on
```

The string would be sent back to the server as one line, not three. If a *CHECKBOX* is not selected, it is omitted from the string. It is up to the server to make sense of this string.

Fortunately, a standard for handling forms' data is already available. It is called **CGI** (**Common Gateway Interface**). Let us consider a common way of

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 28

using it. Suppose that someone has an interesting database (e.g., an index of Web pages by keyword and topic) and wants to make it available to Web users. The CGI way to make the database available is to write a script (or program) that interfaces (i.e., gateways) between the database and the Web. This script is given a URL, by convention in the directory *cgi-bin*. HTTP servers know (or can be told) that when they have to invoke a method on a page located in *cgi-bin*, they are to interpret the file name as being an executable script or program and start it up.

Eventually, some user opens the form associated with our widget script and has it displayed. After the form has been filled out, the user clicks on the SUBMIT button. This action causes the browser to establish a TCP connection to the URL listed in the form's *ACTION* parameter—the script in the *cgi-bin* directory. Then the browser invokes the operation specified by the form's *METHOD*, usually *POST*. The result of this operation is that the script is started and presented (via the TCP connection, on standard input) with the long string given above. In addition, several environment variables are set. For example, the environment variable *CONTENT_LENGTH* tells how long the input string is.

At this point, most scripts need to parse their input to put it in a more convenient form. This goal can be accomplished by calling one of the many libraries or script procedures available. The script can then interact with its database in any way it wishes. For example, active maps normally use CGI scripts to take different actions depending on where the user pointed.

CGI scripts can also produce output and do many other things as well as accepting input from forms. If a hyperlink points to a CGI script, when that link is invoked, the script is started up, with several environment variables set to provide some information about the user. The script then writes a file (e.g. an HTML page) on standard output, which is shipped to the browser and interpreted there. This mechanism makes it possible for the script to generate custom Web pages on the spot.

For better or worse, some Web sites that answer queries have a database of advertisements that can be selectively included in the Web page being constructed, depending on what the user is looking for. If the user is searching for "car" a General Motors ad might be displayed, whereas a search for "vacation" might produce an ad from United Airlines. These ads usually include clickable text and pictures.

7.6.4. Java

HTML makes it possible to describe how static Web pages should appear, including tables and pictures. With the cgi-bin hack, it is also possible to have a limited amount of two-way interaction (forms, etc.). However, rapid interaction with Web pages written in HTML is not possible. To make it possible to have

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 29

highly interactive Web pages, a different mechanism is needed. In this section we will describe one such mechanism, the Java™ language and interpreter.

Java originated when some people at Sun Microsystems were trying to develop a new language that was suitable for programming information-oriented consumer appliances. Later it was reoriented toward the World Wide Web. Although Java borrows many ideas and some syntax from C and C++, it is a new object-oriented language, compatible with neither. It is sometimes said that in the large, Java is like Smalltalk, but that in the small it is like C or C++.

The main idea of using Java for interactive Web pages is that a Web page can point to a small Java program, called an **applet** (SAT I verbal analogy question: Pig is to piglet as application is to ?). When the browser reaches it, the applet is downloaded to the client machine and executed there in a secure way. It must be structurally impossible for the applet to read or write any files that it is not authorized to access. It must also be impossible for the applet to introduce viruses or cause any other damage. For these reasons, and to achieve portability across machines, applets are compiled to a bytecode after being written and debugged. It is these bytecode programs that are pointed to by Web pages, similar to the way images are pointed to. When an applet arrives, it is executed interpretively in a secure environment.

Before getting into the details of the Java language, it is worth saying a few words about what the whole Java system is good for and why people want to include Java applets in their Web pages. For one thing, applets allow Web pages to become interactive. For example, a web page can contain a board for tic tac toe, othello, or chess, and play a game with the user. The game-playing program (written in Java) is just downloaded along with its Web page. As a second example, complex forms (e.g., spreadsheets) can be displayed, with the user filling in items and seeing calculations made instantly.

It is entirely possible that in the long run, the model of people buying programs, installing them, and running them locally will be replaced by a model in which people click on Web pages, get applets downloaded to do work for them, possibly in conjunction with a remote server or data base. Instead of filling out the income tax form by hand or using a special program, people may be able to click on the IRS home page to get a tax applet downloaded. This applet might ask some questions, then contact the person's employer, bank, and stockbroker to collect the required salary, interest, and dividend information, fill the tax form in, and then display it for verification and submission.

Another reason for running applets on the client machine is they make it possible to add animation and sound to Web pages without having to spawn external viewers. The sound can be played when the page is loaded, as background music, or when some specific event happens (e.g., clicking on the cat makes it meow). The same is true for animation. Because the applet is running locally, even if it is being interpreted, it can write all over (its portion) of the screen any way it wants to, and at very high speed (compared to a remote cgi-bin shell script).

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 30

The Java system has three parts:

1. A Java-to-bytecode compiler.
2. A browser that understands applets.
3. A bytecode interpreter.

The developer writes the applet in Java, then compiles it to bytecode. To include this compiled applet on a Web page, a new HTML tag, <APPLET>, has been invented. A typical use is

```
<APPLET CODE=game.class WIDTH=100 HEIGHT=200> </APPLET>
```

When the browser sees the <APPLET> tag, it fetches the compiled applet *game.class* from the current Web page's site (or if another parameter, *CODEBASE*, is present, from the URL it specifies). The browser than passes the applet to the local bytecode interpreter for execution (or interprets the applet itself, if it has an internal interpreter). The *WIDTH* and *HEIGHT* parameters give the size of the applet's default window, in pixels.

In a sense, the <APPLET> tag is analogous to the <IMG> tag. In both cases, the browser goes and gets a file and then hands it off to a (possibly internal) interpreter for display within a bounded area of the screen. Then it continues processing the Web page.

For applications that need very high performance, some Java interpreters have the ability to compile bytecode programs to actual machine language on-the-fly, as needed.

As a consequence of this model, Java-based browsers are extensible in a way that first-generation browsers are not. First generation browsers are basically HTML interpreters that have built-in modules for speaking the various protocols needed, such as HTTP 1.0, FTP, etc., as well as decoders for various image formats. An example is shown in Fig. 7-68(a). If someone invents or popularizes a new format, such as audio or MPEG-2, these old browsers are not able to read pages containing them. At best, the user has to find, download, and install an appropriate external viewer.

With a Java-based browser, the situation is different. At startup, the browser is effectively an empty Java virtual machine, as shown in Fig. 7-68(b). By loading HTML and HTTP applets, it becomes able to read standard Web pages. However, as new protocols and decoders are required, their classes are loaded dynamically, possibly over the network from sites specified in Web pages. After a while, the browser might look like Fig. 7-68(c).

Thus if someone invents a new format, all that person has to do is include the URL of an applet for handling it in a Web page, and the browser will automatically fetch and load the applet. No first-generation browser is capable of automatically downloading and installing new external viewers on-the-fly. The ability to

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 31

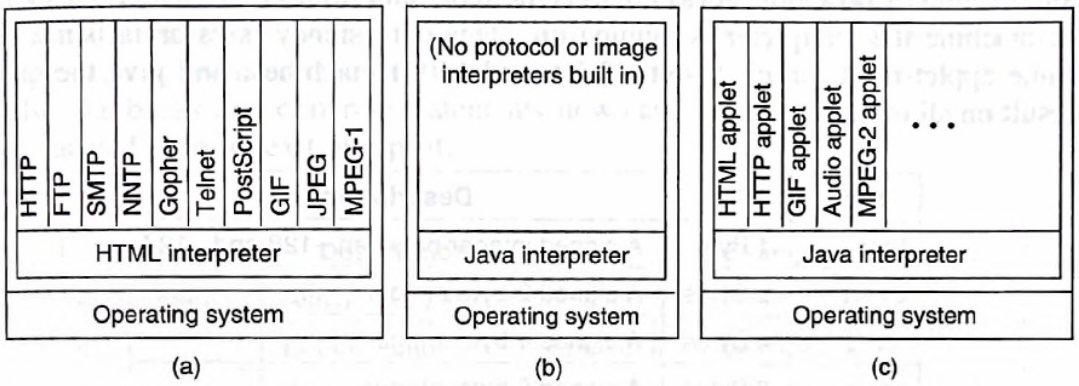


Fig. 7-68. (a) A first generation browser. (b) A Java-based browser at startup. (c) The browser of (b) after running for a while.

load applets dynamically means that people can easily experiment with new formats without first having to have endless standardization meetings to reach a consensus.

This extensibility also applies to protocols. For some applications, special protocols are needed, for example, secure protocols for banking and commerce. With Java, these protocols can be loaded dynamically as needed, and there is no need to achieve universal standardization. To communicate with company X, you just download its protocol applet. To talk to company Y, you get its protocol applet. There is no need for X and Y to agree on a standard protocol.

Introduction to the Java Language

The objectives listed above have led to a type-safe, object-oriented language with built-in multithreading and no undefined or system dependent features. What follows is a highly simplified description of Java, just to give a feel for it. Many features, details, options, and special cases have been omitted for the sake of brevity. The complete language specification, and much more about Java, is available on the Web itself (naturally) at *http://java.sun.com*. For tutorials on Java, see (Campione and Walrath, 1996; and Van der Linden, 1996). For the full story, see (Arnold and Gosling, 1996; and Gosling et al., 1996). For a brief comparison between Java and Microsoft's answer to it (Blackbird), see (Singleton, 1996).

As we mentioned above, in the small, Java is similar to C and C++. The lexical rules, for example, are pretty much the same (e.g., tokens are delimited by white space, and new lines can be inserted between any two tokens). Comments can be inserted using either the C syntax (/* ... */) or the C++ syntax (// ...).

Java has eight primitive data types, as listed in Fig. 7-69. Each type has a specific size, independent of the local implementation. Thus unlike C, where an integer may be 16, 32, or 64 bits, depending on the underlying machine

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 32

architecture, a Java int is always 32 bits, no more and no less, no matter what kind of machine the interpreter is running on. This consistency is essential since the same applet must run on 16-bit, 32-bit, and 64-bit machines, and give the same result on all of them.

Type	Size	Description
Byte	1 Byte	A signed integer between −128 and +127
Short	2 Bytes	A signed 2-byte integer
Int	4 Bytes	A signed 4-byte integer
Long	8 Bytes	A signed 8-byte integer
Float	4 Bytes	A 4-byte IEEE floating-point number
Double	8 Bytes	An 8-byte IEEE floating-point number
Boolean	1 Bit	The only values are true and false
Char	2 Bytes	A character in Unicode

Fig. 7-69. The basic Java data types.

Arithmetic variables (the first 6 types) can be combined using the usual arithmetic operators (including ++ and −−) and compared using the usual relational operators (e.g., <, <=, ==, !=). Conversions between types are permitted where they make sense.

Java uses the 16-bit Unicode instead of ASCII for characters, so character variables are 2 bytes long. The first 127 Unicode characters are the same as ASCII for backward compatibility. Above these are some graphic symbols, and then the characters needed for Russian, Arabic, Hebrew, Japanese (kanji, katakana, and hiragana), and virtually every other language. Characters not present in ASCII can be represented with \u followed by four hexadecimal digits. For example, \u0ae6 is the Gujarati zero.

Java allows one dimensional arrays to be declared. For example,

```
int[] table;
```

declares an array, *table*, but does not allocate any space for it. That can be done later on, as in C++, for example, by

```
table = new int [1024];
```

to allocate an array with 1024 entries. It is not necessary (or even possible) to return arrays that are no longer needed; the garbage collector reclaims them. Thus the highly error-prone *malloc* and *free* library routines are not needed for storage management. Arrays can be initialized, and arrays of arrays can be used to get higher dimensionality, as in C. Strings are available, but they are defined in a class, rather than being simply character arrays ending with a null byte.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 33

The Java control statements are shown in Fig. 7-70. The first nine have essentially the same syntax and semantics as in C, except that where a Boolean expression is required, the language actually insists upon a Boolean expression. Also, the break and continue statements now can take labels indicating which of the labeled loops to exit or repeat.

Statement	Description	Example
Assignment	Assign a value	n = i + j;
If	Boolean choice	if (k < 0) k = 0; else k = 2*k;
Switch	Select a case	switch (b) {case 1: n++; case 2: n—;}
For	Iteration	for (i = 0; i < n; i++) a[i] = b[i];
While	Repetition	while (n < k) n += i;
Do	Repetition	do {n = n + n} while (n < m);
Break	Exit statement	break label;
Return	Return	return n;
Continue	Next iteration	continue label;
Throw	Raise exception	throw new IllegalArgumentException();
Try	Exception scoping	try { ... } catch (Exception e) {return –1};
Synchronized	Mutual exclusion	synchronized void update(int s) { ... }

Fig. 7-70. The Java statements. The notation { ... } indicates a block of code.

The next two statements are in C++ but not in C. The throw and try statements deal with exception handling. Java defines a variety of standard exceptions, such as attempting to divide by zero, and allows programmers to define and raise their own exceptions. Programmers can write handlers to catch exceptions, making it unnecessary to constantly test if something has gone wrong (e.g., when reading from a file). The throw statement raises an exception, and the try statement defines a scope to associate exception handlers with a block of code in which an exception might occur.

The synchronized statement is new to Java and has to do with the fact that Java programs can have multiple threads of control. To avoid race conditions, this statement is used to delimit a block of code (or a whole procedure) that must not have more than one thread active in it at once. Such blocks of code are usually called **critical regions**. When the synchronized statement is executed, the thread executing it must acquire the lock associated with the critical region, execute the code, and then release the lock. If the lock is not available, the thread waits until it is free. By guarding entire procedures this way and using condition variables, programmers have the full power of monitors (Hoare, 1974).

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 34

Java programs can be called with arguments. Command-line processing is similar to C, except that the argument array is called *args* instead of *argv* and *args*[0] is the first parameter, not the program name. Figure 7-71 illustrates a small Java program that computes a table of factorials, just to give an idea of what a small Java program looks like.

```
class Factorial { /* This program consists of a single class with two methods. */

  public static void main (int argc, String args[]) { // main program
  long i, f, lower = 1, upper = 20;          // declarations of four longs

  for (i = lower; i <= upper; i++) {         // loop from lower to upper
      f = factorial(i);                      // f = i!
      System.out.println(i + " " + f);       // print i and f
  }
}

  static long factorial (long k) {           // recursive factorial function
      if (k == 0)
         return 1;                           // 0! = 1
      else
         return k * factorial(k-1);          // k! = k * (k-1)!
  }
}
```

Fig. 7-71. A Java program for computing and printing 0! to 20!.

Despite both being object-oriented languages based on C, Java and C++ differ in some ways. Some features were removed from Java to make it typesafe or easier to read. These include #define, typedef, enums, unions, structs, operator overloading, explicit pointers, global variables, standalone functions, and friend functions. It almost goes without saying that the goto statement has been sent to that special place reserved for obsolete programming language features. Other features were added to give the language more power. The features added include garbage collection, multithreading, object interfaces, and packages.

Object Orientation in Java

In traditional procedural languages such as Pascal or C, a program consists of a collection of variables and procedures, without any general organizing principle. In contrast in **object-oriented languages**, (almost) everything is an object. An **object** normally contains some internal (i.e., hidden) state variables along with some public procedures, called **methods**, for accessing them. Programs that use the object are expected (and can be forced) to invoke the methods to manipulate the object's state. In this way, the object writer can control how programs use the

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 35

information inside the object. This principle is called **encapsulation**, and is the basis of all object-oriented programming.

Java tries to capture the best of both worlds. It can be used as a traditional procedural language or as an object-oriented language. The Java example of Fig. 7-71, for example, could equally well have been written in C, and in essentially the same way. In this view, a subset of Java can be regarded as a cleaned-up version of C. However, for writing Web pages, Java is better regarded as an object-oriented language, so we will study its object orientation in this section.

A Java program consists of one or more **packages**, each of which contains some class definitions. Packages can be accessed remotely over a network, so those intended for use by a wide audience must have unique names. Normally, hierarchical names are used, starting with the reverse of their machine's DNS name, for example

```
EDU.univ.cs.catie.games.chess
```

A **class** definition is a template for stamping out object instances, each of which contains the same state variables and same methods as all the other object instances of its class. The values of the state variables within different objects are independent, however. Classes are thus like cookie cutters: they are not cookies themselves, but are used to stamp out structurally identical cookies, with each cookie cutter producing a different shape of cookie. Once produced, different cookies (objects) are independent of one another.

Java objects can be produced dynamically during execution, for example by

```
object = new ClassName()
```

These objects are stored on the heap and removed by the garbage collector when no longer needed. In this way, storage management in Java is handled by the system, with no need for the dreaded *malloc* and *free* procedures, or even for explicit pointers, for that matter.

Each class is based on another class. A newly defined class is said to be a **subclass** of the class on which it is based, the **superclass**. A (sub)class always **inherits** the methods of its superclass. It may or may not have direct access to the superclass' internal variables, depending on whether or not the superclass wants that. For example, if a superclass, *A*, has methods *M1*, *M2*, and *M3*, and a subclass, *B*, defines a new method, *M4*, then objects created from *B*, will have methods *M1*, *M2*, *M3*, and *M4*. The property of a class automatically acquiring all the methods of its superclass is called **inheritance**, and is an important property of Java. Adding new methods to the superclass' methods is called **extending** the superclass. As an aside, some object-oriented languages allow classes to inherit methods from two or more superclasses (multiple inheritance), but the Java designers thought this property to be too messy and intentionally left it out.

Since every class has exactly one immediate superclass, the set of all classes in a Java program form a tree. The class at the top of the tree is called **Object**.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 36

All other classes inherit its methods. Any class whose superclass is not explicitly mentioned in its definition defaults to being a subclass of the *Object* class. The *Factorial* class of Fig. 7-71, for example, is thus a subclass of *Object*.

Let us now take a look at an example of the object-oriented concepts presented so far. In Fig. 7-72 we have a package defining two classes, *ComplexNumber*, for defining and using complex numbers (i.e., numbers with a real part and an imaginary part), and *test*, for showing how *ComplexNumber* can be used.

```
class ComplexNumber {          // Define a subclass of Object called ComplexNumber
  // Hidden data.
  protected double re, im;     // real and imaginary parts

  // Five methods that manage the hidden data.
  public void Complex(double x, double y) {re = x; im = y;}
  public double Real() {return re;}
  public double Imaginary() {return im;}
  public double Magnitude() {return Math.sqrt(re*re + im*im);}
  public double Angle() {return Math.atan(im/re);}
}

class test {                   // A second class, for testing ComplexNumber
  public static void main (String args[]) {
    ComplexNumber c;           // declare an object of class ComplexNumber

    c = new ComplexNumber();   // actually allocate storage for c
    c.Complex(3.0, 4.0);       // invoke the Complex method to initialize c
    System.out.println("The magnitude of c is " + c.Magnitude() );
  }
}
```

Fig. 7-72. A package defining two classes.

Like *Factorial*, the class *ComplexNumber* is based on *Object*, because no other superclass is named in its definition. Each object of class *ComplexNumber* represents one complex number. Each object of this class contains two hidden variables, *re*, and *im*, both 64-bit floating-point numbers, for representing the real and imaginary parts, respectively. They cannot be accessed outside the class definition (and its subclasses), because they have been declared protected. Had they been declared private, then they would have been visible only to *ComplexNumber* and not to any subclasses. For the moment, private would have been fine, but we will soon define a subclass. Had they been declared public, they would have been visible everywhere the package was visible, thus destroying much of the value of object-oriented programming. Nevertheless, situations do exist in which having the internal state of an object be public is sometimes needed.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 37

Five methods are defined on objects belonging to class *ComplexNumber*. Users of the class are thus restricted to the operations provided by these five methods, and cannot get at the state directly. An example of how objects of class *ComplexNumber* are created, initialized, and used is given in *test*.

When this package is compiled, the Java compiler produces two binary (bytecode) files, one containing each of the classes and named after its class. Typing the command

```
java test
```

results in invoking the Java interpreter with class *test* as parameter. The interpreter then looks for a method called *main*, and upon finding it, executes it. The result of execution is that the line

```
The magnitude of c is 5
```

is printed out.

Now let us define a subclass of *ComplexNumber*, just to see how that works. It starts out by importing the original class, to learn what it does and what its methods are. Then it defines an extension of *ComplexNumber*, which we will call *HairyNumber*. The new class automatically inherits the five methods present in the superclass. To make things interesting, we will define a sixth method, *AddTo*, in the subclass, which adds a complex number to the object, increasing its real and imaginary parts.

The subclass definition is shown in Fig. 7-73, along with another test program showing how an object belonging to class *HairyNumber* can be used. When the new test program is run, it will print out

```
h = (-0.5,6)
```

Remember that the six methods are usable on the objects *a* and *h*, without regard to which method was defined where. If we now define yet another subclass based on *HairyNumber* and give it, say, three new methods, objects produced from it will have nine valid methods.

In addition to adding new methods to its superclass, a subclass can override (replace) existing methods by simply redefining them. Thus it is possible for a subclass to redefine all the methods inherited from its superclass, so objects belonging to the two classes have nothing in common. Doing so, however, is in poor taste, and should be avoided.

Finally, a Java class may define multiple methods with the same name but different parameters and different definitions. When the compiler sees a method invocation using this name, it has to use the parameter types to determine which method to use. This property is called **overloading** or **polymorphism**. Unlike C++, where operators can also be overloaded, in Java, only methods, not operators, can be overloaded, to make programs easier to understand.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 38

```
import ComplexNumber;          // import the ComplexNumber package

class HairyNumber extends ComplexNumber {  // define a new class
  public void AddTo(ComplexNumber z) {     // with one method
     re = re + z.Real();
     im = im + z.Imaginary();
  }
}

class test2 {                  // test program for HairyNumber
  public static void main(String args[]) {
    HairyNumber a, h;          // declare two HairyNumbers

    a = new HairyNumber();     // allocate storage for a
    h = new HairyNumber();     // allocate storage for h
    a.Complex(1.0, 2.0);       // assign a value to a
    h.Complex(-1.5, 4.0);      // assign a value to h
    h.AddTo(a);                // invoke the AddTo method on h
    System.out.println("h = (" + h.Real() + "," + h.Imaginary() + ")" );
  }
}
```

Fig. 7-73. A subclass of *ComplexNumber* defining a new method.

The Application Programmers Interface

In addition to the bare language, the Java designers have defined and implemented about 200 classes with the initial release. The methods contained in these classes form a kind of standard environment for Java program developers. The classes are written in Java, so they are portable to all platforms and operating systems.

While a detailed discussion of all these classes and methods is clearly beyond the scope of this book, a brief description may be of some interest. The 200 classes are grouped into seven packages of uneven size, each of which is focused on some central theme. Applets that need a particular package can include it using the Java import statement. The methods contained within can just be used as needed. This mechanism replaces the need for including header files in C. It also replaces the need for libraries, since the packages are dynamically loaded during execution when they are invoked.

The seven packages are summarized in Fig. 7-74. The *java.lang* package contains classes that can be viewed as part of the language, but are technically not. These include classes for managing the classes themselves, threads, and exception handling. The standard mathematical and string libraries are also here.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 39

Package	Example functionality
Java.lang	Classes, threads, exceptions, math, strings
Java.io	I/O on streams and random access files, printing
Java.net	Sockets, IP addresses, URLs, datagrams
Java.util	Stacks, hash tables, vectors, time, date
Java.applet	Getting and displaying Web pages, audio, Object class
Java.awt	Events, dialog, menus, fonts, graphics, window management
Java.awt.image	Colors, image cropping, filtering, and conversion
Java.awt.peer	Access to the underlying window system

Fig. 7-74. The packages included in the standard API.

Like C, the Java language contains no I/O primitives. I/O is done by loading and using the *java.io* package. It is analogous to the standard I/O library in C. Methods are provided for reading and writing streams, random access files, and doing the formatting needed for printing. In Fig. 7-71 we saw one of these methods, *println*, which does formatted printing.

Closely related to I/O is network transport. Methods that look up and manage IP addresses are located here. Access to sockets is also part of this package. So is datagram preparation. The actual transmission is handled in *java.io*.

The next package is *java.util*. It contains classes and methods for common data structures, such as stacks and hash tables, so programmers do not constantly have to reinvent the wheel. Time and date management is also here.

The *java.applet* package contains some of the basic machinery for applets, including methods for getting Web pages starting from their URLs. It also has methods for displaying Web pages and playing audio clips (e.g., background music). The *java.applet* package also contains the *Object* class. All objects inherit its methods, unless they are overridden. These methods include cloning an object, comparing two objects for equality, converting an object to a string, and various others.

Finally, we come to *java.awt* and its two subpackages. AWT stands for **Abstract Window Toolkit**, and is designed to make applets portable across window systems. For example, how should an applet draw a rectangle on the screen in such a way that the same compiled (bytecode) version of the applet can run on UNIX, Windows, and the Macintosh, even though each one has its own window system? Part of the package deals with drawing on the screen, so there are methods for placing lines, geometric figures, text, menus, buttons, scroll bars, and many other items on the screen. Java programmers call these methods to write on the screen. It is up to the *java.awt* package to make the appropriate calls to the local operating system to get the job done. This strategy means that *java.awt* has

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 40

to be rewritten for each new platform, but that applets are then platform independent, which is far more important.

Another important task of this class is event management. Most window systems are fundamentally event driven. What this means is that the operating system detects keystrokes, mouse motion, button pushes and releases, and other events, and converts these into calls to user procedures. In the case of Java, a large library of methods for dealing with these events is provided in *java.awt*. Using them makes it easier to write programs that interact with the local window system and still be 100 percent portable to machines with different operating systems and different window systems.

Some of the work of this package is done in *java.awt.image*, such as image management, and in *java.awt.peer*, which allows access to the underlying window system.

Security

One of the most important aspects of Java is its security properties. When a Web page containing an applet is fetched, the applet is automatically executed on the client's machine. Ideally, it should not crash or otherwise bring down the client's machine.

Furthermore, it does not take much imagination to envision some enterprising undergraduate producing a Web page containing some nifty new game, then publicizing its URL widely (e.g., crossposting it to every newsgroup). Not mentioned in the posting is the small detail that the page also contains an applet that upon arrival immediately encrypts all the files on the user's hard disk. When it is finished, the applet announces what it has done and politely mentions that users wishing to purchase the decryption key can do so by sending 1000 dollars in small unmarked bills to a certain post office box in Panama.

In addition to the above get-rich-quick scheme, there are other dangers inherent in letting foreign code run on your machine. An applet could hunt around for interesting information (saved email, the password file, the local environment strings, etc.) and send or email them back over the network. It could also consume resources (e.g., filling up the disk), display naughty pictures or political slogans on the screen, or make an earsplitting racket using the sound card.

The Java designers were well aware of these problems, of course, and erected a series of barriers against them. The first line of defense is a typesafe language. Java has strong typing, true arrays with bounds checking and no pointers. These restrictions make it impossible for a Java program to construct a pointer to read and write arbitrary memory locations.

However, Trudy, who has given up on trying to break cryptographic protocols and gotten into the much more interesting business of writing malicious Java applets, can just write or modify a C compiler to produce Java bytecode, thus bypassing all the safeguards provided by the Java language and compiler.

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 41

The second line of defense is that before an incoming applet is executed, it is run through a bytecode verifier. The bytecode verifier looks for attempts to manufacture pointers, execute instructions or call methods with invalid parameters, use variables before they are initialized, and so on. These checks are supposed to guarantee that only legal applets get executed, but Trudy will certainly work hard on finding tricks the verifier does not check for.

The third line of defense is the class loader. Since classes can be loaded on the fly, there is a danger that an applet could load one of its own classes to replace a critical system class, thus bypassing that class' security checks. This Trojan horse attack has been rendered impossible by virtue of giving each class its own name space (like a kind of abstract directory), and carefully searching for system classes before looking for user classes. In other words, if the user loads a malicious version of *println*, it will never be used because the official *println* will always be found first.

The fourth line of defense is that some standard classes have their own security measures built in. For example, the file access class maintains a list of files that may be accessed by applets, and pops up a dialog box any time an applet tries to do something that violates the protection rules.

Despite all these measures, security problems are to be expected. First, there can be bugs in the Java software that clever programmers can exploit to bypass the security. The infamous Internet worm of 1988 used a bug in the UNIX Finger daemon to bring thousands of machines all over the Internet to a grinding halt (Hafner and Markoff, 1991; and Spafford, 1989).

Second, while it may be possible to prevent an applet from doing anything except writing to the screen, many applets will need more power, so when they ask for additional privileges, users may grudgingly (or naively) grant them. For example, applets may need to write temporary files, so users may give them access to the */tmp* directory, thinking that nothing important is there. Unfortunately, most editors keep the temporary versions of documents and email being edited there, so malicious applets can copy them and try to send them over the network. Of course, it may be possible to block applets' access to the network, but many may not work then, so they will have to be granted this power too.

But even in the unlikely event that applets are allowed no network access at all, they may be able to transmit information using **covert channels** (Lampson, 1973). For example, after acquiring some information, an applet can form a bit stream by using the local system's real time clock. To send a 1, it computes very hard for ΔT; to send a 0, it just waits for ΔT.

To acquire this information, the applet's owner can establish a connection to the client's machine to read some of its public Web pages or FTP some of its public files. By carefully monitoring the incoming data rate, the applet owner's can see whether the applet is computing (and thus slowing down the observed output stream) or resting. Of course, this channel is noisy, but that can be handled by standard techniques. The stream can be divided into frames delimited by flag

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 42

bytes, individual frames can use a strong error-correcting code, and all frames can be sent two or three times. Many other covert channels exist, and it is extremely difficult to discover and block them all. For more information about the security problems in Java see (Dean and Wallach, 1995).

In short, Java introduces many new possibilities and opportunities into the World Wide Web. It allows Web pages to be interactive, and to contain animation and sound. It also permits browsers to be infinitely extensible. However, the Java model of downloading applets also introduces some serious new security problems that have not been entirely solved yet.

7.6.5. Locating Information on the Web

Although the Web contains a vast amount of information, finding the right item is not always easy. To make it easier for people to find pages that are useful to them, several researchers have written programs to index the Web in various ways. Some of these have become so popular that they have gone commercial. Programs that search the Web are sometimes called **search engines**, **spiders**, **crawlers**, **worms**, or **knowbots** (knowledge robots). In this section we will give a brief introduction to this subject. For more information, see (Pinkerton, 1994; and McBryan, 1994).

Although the Web is huge, reduced to its barest essentials, the Web is a big graph, with the pages being the nodes and the hyperlinks being the arcs. Algorithms for visiting all the nodes in a graph are well known. What makes Web indexing difficult is the enormous amount of data that must be managed and the fact that it is constantly changing.

Let us start our discussion with a simple goal: indexing all the keywords in Web pages' titles. For our algorithm, we will need three data structures. First, we need a large, linear array, *url_table*, that contains millions of entries, ultimately one per Web page. It should be kept in virtual memory, so parts not heavily used will automatically be paged to disk. Each entry contains two pointers, one to the page's URL and one to the page's title. Both of these items are variable length strings and can be kept on a heap (a large unstructured chunk of virtual memory to which strings can be appended). The heap is our second data structure.

The third data structure is a hash table of size n entries. It is used as follows. Any URL can be run through a hash function to produce a nonnegative integer less than n. All URLs that hash to the value k are hooked together on a linked list starting at entry k of the hash table. Whenever a URL is entered into *url_table*, it is also entered into the hash table. The main use of the hash table is to start with a URL and be able to quickly determine whether it is already present in *url_table*. These three data structures are illustrated in Fig. 7-75.

Building the index requires two phases: searching and indexing. Let us start with a simple engine for doing the searching. The heart of the search engine is a recursive procedure *process_url*, which takes a URL string as input. It operates as

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 43

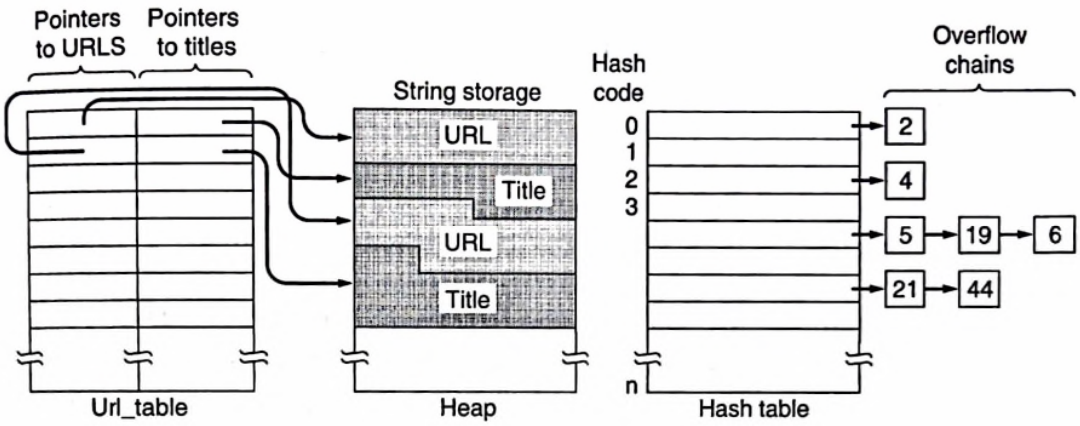


Fig. 7-75. Data structures used in a simple search engine.

follows. First, it hashes the URL to see if it is already present in *url_table*. If so, it is done and returns immediately. Each URL is processed only once.

If the URL is not already known, its page is fetched. The URL and title are then copied to the heap and pointers to these two strings are entered in *url_table*. The URL is also entered into the hash table.

Finally, *process_url* extracts all the hyperlinks from the page and calls *process_url* once per hyperlink, passing the hyperlink's URL as the input parameter.

To run the search engine, *process_url* is called with some starting URL. When it returns, all pages reachable from that URL have been entered into *url_table* and the search phase has been completed.

Although this design is simple and theoretically correct, it has a serious problem in a system as large as the Web. The problem is that this algorithm does a depth-first search, and will ultimately go into recursion as many times as the longest noncyclic path on the Web. No one knows how long this path is, but it is probably thousands of hyperlinks long. As a consequence, any search engine that uses this depth-first search will probably hit stack overflow before finishing the job.

In practice, actual search engines first collect all the hyperlinks on each page they read, remove all the ones that have already been processed, and save the rest. The Web is then searched breadth-first; that is, each link on a page is followed and all the hyperlinks on all the pages pointed to are collected, but they are not traced in the order obtained.

The second phase does the keyword indexing. The indexing procedure goes down *url_table* linearly, processing each entry in turn. For each entry, it examines the title and selects out all words not on the stop list. (The stop list prevents indexing of prepositions, conjunctions, articles, and other words with many hits and little value.) For each word selected, it writes a line consisting of the word

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 44

followed by the current *url_table* entry number to a file. When the whole table has been scanned, the file is sorted by word.

The index will have to be stored on disk and can be used as follows. The user fills in a form listing one or more keywords and clicks on the SUBMIT button. This action causes a *POST* request to be done to a CGI script on the machine where the index is located. This script (or, more likely, program) then looks up the keywords in the index to find the set of *url_table* indices for each one. If the user wants the BOOLEAN AND of the keywords, the set intersection is computed. If the BOOLEAN OR is desired, the set union is computed.

The script now indexes into *url_table* to find all the titles and URLs. These are then combined to form a Web page and are sent back to the user as the response to the *POST*. The browser now displays the page, allowing the user to click on any items that appear interesting.

Sounds easy? It's not. The following problems have to be solved in any practical system:

1. Some URLs are obsolete (i.e., point to pages that no longer exist).
2. Some machines will be temporarily unreachable.
3. Not all pages may be reachable from the starting URL.
4. Some pages may be reachable only from active maps.
5. Some documents cannot be indexed (e.g., audio clips).
6. Not all documents have (useful) titles.
7. The search engine could run out of memory or disk space.
8. The entire process might take too long.

Obsolete URLs waste time but are mostly a nuisance because the server on which they are supposed to be located replies immediately with an error code. In contrast, when the server is down, all the search engine observes is a long delay in establishing the TCP connection. To prevent it from hanging indefinitely, it must have a timeout. If the timeout is too short, valid URLs will be missed. If it is too long, searching will be slowed down appreciably.

Choosing the starting URL is clearly a major issue. If the search engine starts with the home page of some astrophysicist, it may eventually find everything on astronomy, physics, chemistry and space science, but it may miss pages about veterinary medicine, Middle English, and rock 'n roll completely. These sets may simply be disjoint. One solution is to gather as large a set of URLs as possible, and use each of them as a starting page. Starting URLs can be gathered from USENET news articles and last week's version of the *url_table*, since some of these pages may have changed recently (e.g., one of the astrophysicists married a veterinarian and they solemnly updated their home pages to point to each other).

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 45

Indexing works well on text, but increasingly, many pages contain items other than text, including pictures, audio, and video. One approach here is to probe each new-found URL with the *HEAD* method, just to get back its MIME header. Anything not of type *text* is not searched.

About 20 percent of all Web pages have no title, and many of those that do have a title have a quasi-useless one ("Joe's page"). A big improvement to the basic index is to not only include titles, but also all the hypertext. In this approach, when a page is scanned, all the hyperlinks are also recorded, along with the page they came from and the page they point to. After the search phase has been completed, all the hyperwords can be indexed too.

Even more ambitious is to index all the important words in each page. To determine the important words, the occurrence frequency of all words not on the stop list can be computed (per Web page). The top 10 or 20 words are probably worth indexing. After all, if the word "liver" is the most common word on a page, there is a chance that the page will be of interest to biliary surgeons (or to cooks). Some search engines (e.g., Lycos) use this strategy.

Finally, the search engine can run out of memory or time. One attack is to redesign the algorithms more carefully. A completely different approach is to do what Harvest does and offload the work (Bowman et al., 1994, 1996). In particular, Harvest provides a program to run on cooperating servers. This program does all the searching locally and transmits back the finished local index. At the central site, all the local indices are merged into the master index. This approach reduces by orders of magnitude the amount of memory, CPU time, and network bandwidth required but has the major disadvantage of requiring all Web servers to cooperate by running foreign software. Given the potential problems with viruses and worms, when a system administrator is approached with the request: "Will you please run this program on your machine for me?" it should not be surprising if many of them decline.

One small request is in order. Although writing a search engine sounds easy, a buggy one can wreak havoc with the network by generating vast numbers of spurious requests, not only wasting bandwidth but bringing many servers to their knees due to the load. If you cannot resist the temptation to write your own search engine, proper netiquette requires restricting it to your own local DNS domain until it is totally debugged.

7.7. MULTIMEDIA

Multimedia is the holy grail of networking. When the word is mentioned, both the propeller heads and the suits begin salivating as if on cue. The former see immense technical challenges in providing (interactive) video on demand to every home. The latter see equally immense profits in it. No book on networking would be complete without at least an introduction to the subject. Given the

LENNOX EXHIBIT 1015
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00716, Page 46